A Compact and Scalable Hardware/Software Co-design of SIKE

Pedro Maat C. Massolino[1], Patrick Longa[2], Joost Renes[1] and Lejla Batina[1]

[1] Radboud University, Nijmegen, The Netherlands
{p.massolino,j.renes,lejla}@cs.ru.nl
[2] Microsoft Research, USA
plonga@microsoft.com

Abstract. We present efficient and compact hardware/software co-design implementations of the Supersingular Isogeny Key Encapsulation (SIKE) protocol on field-programmable gate arrays (FPGAs). In order to be better equipped for different post-quantum scenarios, our architectures were designed to feature high-flexibility by covering *all* the currently available parameter sets and with support for primes up to 1016 bits. In particular, any of the current SIKE parameters equivalent to the post-quantum security of AES-128/192/256 and SHA3-256 can be selected and run *on-the-fly*. This security scalability property, together with the small footprint and efficiency of our architectures, makes them ideal for embedded applications in a post-quantum world. In addition, the proposed implementations exhibit regular, constant-time execution, which provides protection against timing and simple side-channel attacks. Our results demonstrate that supersingular isogeny-based primitives such as SIDH and SIKE can indeed be deployed for embedded applications featuring competitive performance. For example, our smallest architecture based on a 128-bit MAC unit takes only 3415 slices, 21 BRAMs and 57 DSPs on a Virtex 7 690T and can perform key generation, encapsulation and decapsulation in 14.4, 24.4 and 26.0 milliseconds for SIKEp434 and in 52.3, 86.4 and 93.2 milliseconds for SIKEp751, respectively.

Keywords: Post-quantum cryptography · supersingular isogenies · SIDH · SIKE · hardware/software co-design · FPGA · constant-time · embedded applications

1 Introduction

In 2011, Jao and De Feo introduced a novel key exchange protocol based on the difficulty of computing isogenies between supersingular elliptic curves [JF11]. This key exchange protocol—best known as Supersingular Isogeny Diffie-Hellman (SIDH)—has caught the attention of the cryptographic community, thanks in part to featuring one of the smallest keys of any algorithm known in the literature that is conjectured to be *quantum resistant*. Just in the last few years, there has been a surge of works that look at various aspects of SIDH, including efforts that have focused on the implementation and optimization of this primitive on various platforms [AFJ14, CLN16, KJA$^+$16, KAMK16, SLLH18, SJA19], the introduction of compression techniques that aim at reducing the size of SIDH public keys even further [AJK$^+$16, CJL$^+$17, ZSJP$^+$18, NR19], and the design of other primitives based on SIDH, such as digital signature schemes [JS14, GPS17, YAJ$^+$17] and authenticated key exchange protocols [Gal18, Lon18]. At the same time, the security of this young cryptographic primitive has been analyzed in [BJS14, DG16, GPST16, Pet17].

Recently, the National Institute of Standards and Technology (NIST) launched a multi-year project called “Post-Quantum Cryptography Standardization” [Nat18a] that is aimed at the analysis, selection and standardization of the next-generation of cryptographic algorithms that are believed to be secure against classical *and* quantum computer attacks. Among the 69 final submitters [Nat18b], the so-called Supersingular Isogeny Key Encapsulation (SIKE) protocol [ACC+17] stands out as the candidate with the smallest keys and as the only algorithm that is based on isogeny-based cryptography and, hence, partially relies on traditional Elliptic Curve Cryptography (ECC) arithmetic. SIKE—designed by Costello, De Feo, Jao, Longa, Naehrig and Renes—is an IND-CCA secure key encapsulation mechanism (KEM) based on SIDH which overcomes the inherent vulnerability of the original protocol to the adaptive active attacks by Galbraith, Petit, Shani and Ti [GPST16]. As a consequence, while in the case of SIDH public/secret key pairs should be treated as *ephemeral*, i.e., they must be discarded after use, SIKE key pairs can be freely reused.

The state-of-the-art instantiation of SIDH is based on the work by Costello, Longa and Naehrig in [CLN16], which in turn builds upon the work by De Feo, Jao and Plût [FJP14]. In [ACC+17], this instantiation was used as basis to define the three parameter sets `SIKEp503`, `SIKEp751` and `SIKEp964`, which were named so because of the bit-length of the underlying prime field characteristic. For Round 2 of the NIST process and following recent studies of the classical and quantum security of SIDH and SIKE [ACC+19a, JS19, CLN+19], the parameter set `SIKEp964` was eliminated and two new parameter sets, proposed in [ACC+19a] and called `SIKEp434` and `SIKEp610`, were added [ACC+19b]. One important research effort involves the implementation and optimization of these different parameter sets to evaluate their viability on software and hardware platforms. In this work, we focus on the case of FPGAs as a hardware platform, but presenting our SIKE implementation as hardware/software co-design giving special attention to embedded applications.

1.1 Related work

There exist several hardware implementations of `SIDHp503` and `SIDHp751` [KAMK16, KAMK18, RM19], and of `SIKEp503` and `SIKEp751` [ACC+17, KAK+19]. All those implementations have primarily targeted raw speed as their main goal. This approach typically sacrifices area and, hence, chip cost for performance. However, in embedded applications aiming at low power and energy budgets, area is arguably a more relevant metric. Moreover, the architectures presented in [KAMK16, KAMK18, RM19, KAK+19] can be configured to target only one security level at a time. That is, if the parameter set used in a certain device needs to be changed, the device needs to be reprogrammed, which can be an expensive process that is even unfeasible in certain settings.

1.2 Our contribution and paper organization

In this work we present the first hardware/software co-design supporting the SIKE protocol, with the primary goal of being *compact* and *scalable*. As a result of this approach, the architecture works for all of the four parameter sets defined in [ACC+19b]. This level of scalability allows a device to run *any* of the supported parameter sets without the need of reprogramming it. Hence, given the risk of potential future attacks against post-quantum cryptosystems, the proposed architecture offers an easy and inexpensive solution to upgrade the security of implementations *on-the-fly*. Yet another consequence of this flexibility is that our architecture also represents the *first* hardware implementation of the new parameter sets included in Round 2 of the NIST process, namely, `SIKEp434` and `SIKEp610`.

We present two instances, depending on the bit-size of the multiplier accumulator (MAC) unit that we refer to as `Carmela`. The first and smaller version uses a 128-bit MAC

Table 1: Timing results for all SIKE parameter sets of our architecture based on `Carmela128` and `Carmela256` on the Virtex 7 690T and Artix 7 100T platforms, measured in milliseconds (ms).

	`Carmela128`			`Carmela256`		
Param.	**Key gen.**	**Encaps.**	**Decaps.**	**Key gen.**	**Encaps.**	**Decaps.**
`SIKEp434`	14.4	24.4	26.0	6.9	11.8	12.5
`SIKEp503`	17.1	28.8	30.7	8.2	13.9	14.8
`SIKEp610`	28.6	53.2	54.0	13.8	25.7	26.1
`SIKEp751`	52.3	86.4	93.2	17.7	29.3	31.5

(a) Results on the Virtex 7 690T.

	`Carmela128`			`Carmela256`		
Param.	**Key gen.**	**Encaps.**	**Decaps.**	**Key gen.**	**Encaps.**	**Decaps.**
`SIKEp434`	15.3	26.0	27.6	9.1	15.4	16.4
`SIKEp503`	18.2	30.7	32.7	10.7	18.1	19.3
`SIKEp610`	30.4	56.7	57.4	18.0	33.6	34.0
`SIKEp751`	55.7	92.0	99.2	23.1	38.2	41.1

(b) Results on the Artix 7 100T.

unit (`Carmela128`), while the second relies on a 256-bit unit (`Carmela256`). Both are fully scalable for primes up to 1016 bits and support all SIKE parameter sets, as mentioned above. The main difference between the two is that the instance based on `Carmela128` is aimed at highly-constrained applications, providing a more extreme trade-off between compactness and speed than the one based on `Carmela256`.

The compactness and efficiency of our implementation is illustrated with results on two different FPGA devices, namely the Virtex 7 690T and the Artix 7 100T. On the Virtex 7 690T the `Carmela128`-based architecture takes 3415 slices, 21 BRAMs and 57 DSPs with maximum clock period of 6.57 ns, while the `Carmela256`-based architecture requires 7408 slices, 38 BRAMs and 162 DSPs with maximum clock period of 7.034 ns. The corresponding timings of the operations are summarized in Table 1(a). Similarly, on the Artix 7 100T the `Carmela128`-based architecture takes 3512 slices, 21 BRAMs and 57 DSPs with maximum clock period of 6.993 ns, while the `Carmela256`-based architecture takes 7491 slices, 37 BRAMs and 162 DSPs with maximum clock period of 9.181 ns. The corresponding results are summarized in Table 1(b). For a full comparison to existing literature we refer to Section 4.

To summarize, to the best of our knowledge the hardware implementations of SIKE in the literature target speed and do not offer other attractive features such as on-the-fly security scalability and compactness. This work aims at filling this gap, targeting embedded applications where one might want to update the parameters without the need to change much of the implementation. We envision this being a likely scenario for such a post-quantum primitive. Our results demonstrate that supersingular isogeny-based primitives such as SIDH and SIKE can indeed be deployed for embedded applications featuring competitive performance.

Finally, not only do we describe our architecture and results but, as an additional contribution, we also release the entire project as open source at:

https://github.com/pmassolino/hw-sike.

The paper is organized as follows. Section 2 gives necessary mathematical details on SIKE and some background information on the protocol. Section 3 describes our hardware

$$\begin{tikzcd}
E_0 \arrow[r, "\phi_\ell"] \arrow[d, "\phi_m"'] & (E_\ell, \phi_\ell(P_m), \phi_\ell(Q_m)) \arrow[d, "\phi'_m"] \\
(E_m, \phi_m(P_\ell), \phi_m(Q_\ell)) \arrow[r, "\phi'_\ell"] & E_{m,\ell} \cong E_{\ell,m}
\end{tikzcd}$$

Figure 1: Supersingular isogeny Diffie-Hellman key exchange (SIDH).

architecture, its inner workings and the choices involved in the design. Section 4 gives the results and performance figures for the architecture including a comparison with previous works. We conclude the paper and give some directions for future work in Section 5.

2 Preliminaries

For completeness we first recall the details of SIDH and SIKE. Throughout this document we follow the notation from [ACC+19b].

2.1 SIDH

Let e_2, e_3 be positive integers such that $p = 2^{e_2}3^{e_3}-1$ is prime. Let $E_0/\mathbb{F}_p : y^2 = x^3+6x^2+x$ be an elliptic curve, which is supersingular [Sil09, Theorem V.3.1(a)] as $p \equiv 3 \bmod 4$. Then one can show that $\#E_0(\mathbb{F}_{p^2}) = (p+1)^2$, and in particular $E_0(\mathbb{F}_{p^2})[2^{e_2}] \cong (\mathbb{Z}/2^{e_2}\mathbb{Z})^2$ and $E_0(\mathbb{F}_{p^2})[3^{e_3}] \cong (\mathbb{Z}/3^{e_3}\mathbb{Z})^2$.

Let Alice choose $m \in \{2,3\}$ and fix basis points $P_m, Q_m \in E_0(\mathbb{F}_{p^2})$ such that $E_0(\mathbb{F}_{p^2})[m^{e_m}] = \langle P_m, Q_m\rangle$. As a result, any $sk_m \in \mathbb{Z}/m^{e_m}\mathbb{Z}$ defines a unique subgroup $\langle P_m + [sk_m]Q_m\rangle \subset E_0(\mathbb{F}_{p^2})[m^{e_m}]$ of order m^{e_m}. In turn, this determines (up to isomorphism) an elliptic curve $E_m = E_0/\langle P_m + [sk_m]Q_m\rangle$ and an m^{e_m}-isogeny $\phi_m : E_0 \to E_m$. We then define Alice's secret key to be sk_m while her public key is chosen to be $pk_m = (E_m, \phi_m(P_\ell), \phi_m(Q_\ell))$, where $P_\ell, Q_\ell \in E_0(\mathbb{F}_{p^2})$ are Bob's basis points. Bob proceeds analogously with $\ell \in \{2,3\}$ such that $\ell \neq m$. We define Bob's secret key to be sk_ℓ while his public key is chosen to be $pk_\ell = (E_\ell, \phi_\ell(P_m), \phi_\ell(Q_m))$.

In order to compute a shared secret Alice and Bob then proceed according to Figure 1, where

$$\ker \phi'_m = \langle \phi_\ell(P_m) + [sk_m]\phi_\ell(Q_m)\rangle\,, \quad \ker \phi'_\ell = \langle \phi_m(P_\ell) + [sk_\ell]\phi_m(Q_\ell)\rangle\,.$$

Correctness follows from the fact that $E_{\ell,m}$ and $E_{m,\ell}$ are both the co-domain of the isogeny from E_0 with kernel $\langle P_m + [sk_m]Q_m, P_\ell + [sk_\ell]Q_\ell\rangle$, which is unique up to post-composition with an isomorphism [Sil09, Exercise III.3.13(e)]. Therefore $K = j(E_{\ell,m}) = j(E_{m,\ell})$ can be used as the shared secret.

The resulting shared key K is indistinguishable from random under a chosen plaintext attack (IND-CPA) assuming the SSDDH problem [FJP14, Problem 5.4] is hard. However, there is no active security as demonstrated by the adaptive attacks by Galbraith et al. [GPST16] that allow to recover the secret key. As a result, one can only securely exchange keys with ephemeral public keys.

2.2 SIKE

To obtain active security one can apply a standard transformation to transform the scheme into an IND-CCA secure key encapsulation mechanism (KEM) referred to as SIKE [ACC+19b]. This is achieved in two steps; first, one transforms the IND-CPA secure key exchange into an IND-CPA secure public-key encryption (PKE) using hashed

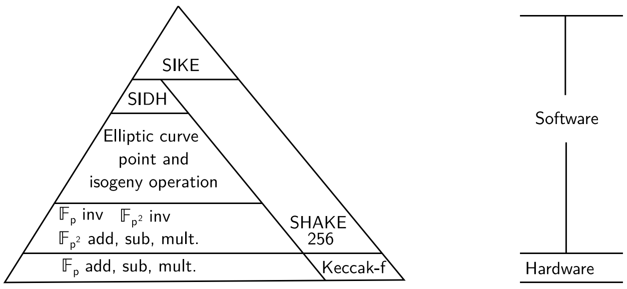
SIKE
SIDH
Elliptic curve point and isogeny operation
$\mathbb{F}_p$ inv $\mathbb{F}_{p^2}$ inv
$\mathbb{F}_{p^2}$ add, sub, mult.
$\mathbb{F}_p$ add, sub, mult.
SHAKE 256
Keccak-f
Software
Hardware

Figure 2: The high-level procedural hierarchy of the SIKE processor.

ElGamal [ACC+19b, Proposition 1], after which one applies a transformation à la Fujisaki–Okamoto [FO13, HHK17] to obtain a secure KEM indistinguishable under chosen ciphertext attacks (IND-CCA). These transformations do not significantly affect the efficiency of the protocol, and we refer to the SIKE specification [ACC+19b] for more details.

For the purpose of this work we have implemented the parameter sets `SIKEp434`, `SIKEp503`, `SIKEp610` and `SIKEp751` defined in [ACC+19b], and we precisely follow their design decisions. This includes the use of Montgomery representation [CH17, Ren18] and performing all arithmetic on the Kummer line [CLN16]. Again, for details we refer to [ACC+19b, §1.3]. The (optional) methods related to public-key compression are considered out of scope for this work.

3 Proposed Scalable Architecture

The SIKE key encapsulation protocol is a complex algorithm that involves many different operations. Therefore, we opted for hardware/software co-design, instead of an approach solely consisting of a control unit plus combinational circuitry. In this way, we split the computational aspects of the protocol into hardware and software logic, as shown in Figure 2. By running the software on top of the fast hardware instructions, we gain flexibility and facilitate maintenance and debugging, while keeping an optimized hardware execution. Our approach for the finite field arithmetic is standard for this sort of complex public-key implementation task. Basically, all the low-level $\mathbb{F}_p$ operations (i.e., multiplication, squaring and addition/subtraction) are implemented directly in hardware, while the inversion in $\mathbb{F}_p$ is done in software. All the other operations that are built on top of the $\mathbb{F}_p$ arithmetic (i.e., the $\mathbb{F}_{p^2}$ computations, the elliptic-curve point and isogeny algorithms and the SIDH algorithm itself) are implemented in software. Finally, SIKE can be obtained by applying the same methodology where, additionally, Keccak is implemented in hardware and the necessary SHAKE calls are done in software.

Our architecture, depicted in Figure 3, is split into two big parts. The first part is the multiplier accumulator (MAC) unit, called `Carmela`, and its respective memory (MAC RAM). It is responsible for the instructions related to the $\mathbb{F}_p$ operations, such as multiplication and addition/subtraction. This unit does not have an internal memory for the input and output operands and, therefore, needs the MAC RAM to work. The MAC RAM is composed of two true dual port RAMs, which together can perform two reads and one write, or two writes, in one cycle. As this unit only performs the arithmetic operations, all the control flow instructions are handled by the main unit and its base ALU and base RAM. Thus, `Carmela` behaves like a coprocessor of the main unit.

The second part (i.e., the main unit) includes the base ALU, the base RAM, the RD registers, address resolution, the local bus, the status registers and the program counter. It is a 16-bit signed/unsigned integer processor that can perform direct and conditional jumps, stack push and pop, copy values through the local bus, load values directly into

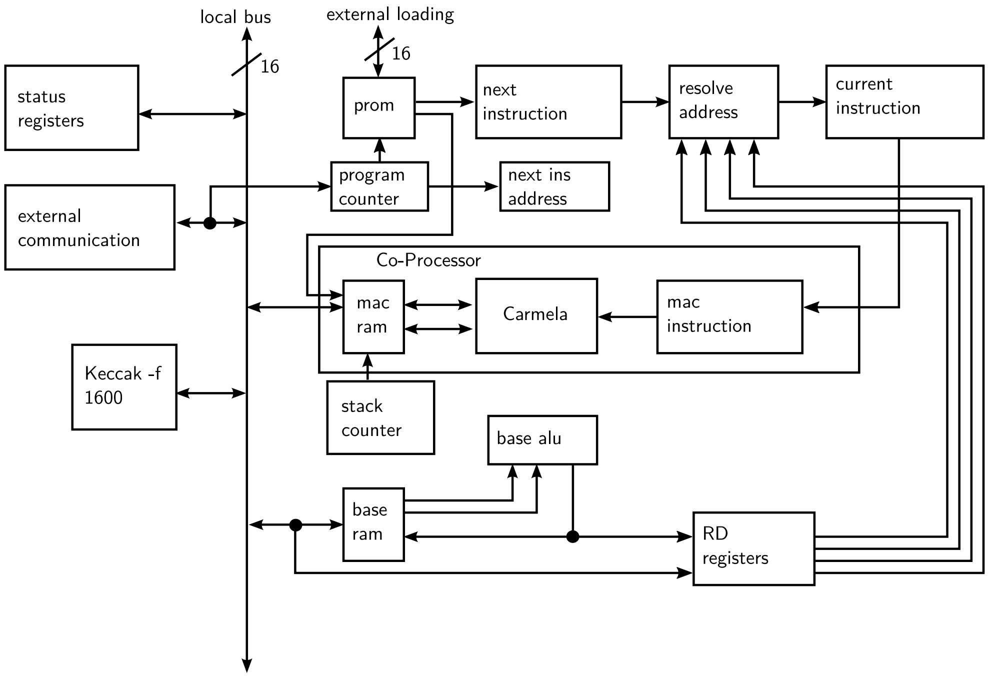

Figure 3: The high-level architecture of the SIKE processor.

the memories, call/return program functions, and execute arithmetic operations such as addition/subtraction and multiplication, and logical operations such as and, not, or, xor. The instruction set was crafted to provide the necessary operations for the SIDH functions, but should also work with other programs or procedures. Future work could involve the change of the instruction set to a well-known and widely adopted ISA, such as RISC-V.

The RD registers unit works as a cache of the first 32 values of the base RAM. This was done because those registers can be read asynchronously, while the base RAM values have 1 cycle of delay. These values are mainly used as memory pointers for the MAC or the main processor operations. Therefore, it is possible to access an operand that is located on the base RAM or the MAC RAM pointed to by the first 32 positions of the base RAM itself. The first RD register has a special purpose (see Figure 4): bit 0 or 15 can be used in conjunction with another pointer register in order to point to a position that is dependent on one of these bits. For example, if there are two pointers stored in position 4 and 5, respectively, the indirect memory access can take the pointer stored in position $(10S)_2$, where S is bit 15 or 0 of the RD register.

The base ALU is the FPGA DSP that can perform addition, subtraction and multiplication. Comparisons are accomplished via the DSP subtraction operation. Finally, the shifts and rotations are done through the mux-based data-reversal barrel shift as showed

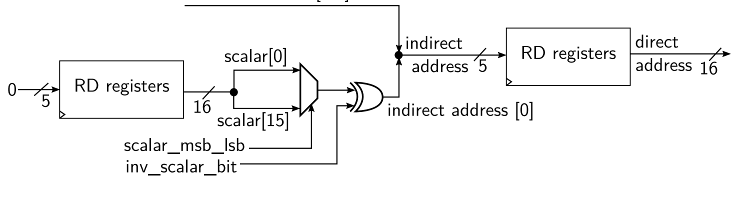

Figure 4: Description of how RD register 0 is used to make an indirect address from one bit and one extra word.

in Pillmeier et al. [PSW02].

The PROM unit holds all the instructions to be run by the main unit and the MAC unit. An instruction first has its address resolved and is then executed by the main or MAC unit, respectively. In case it is a MAC instruction, the main unit serially loads 8 instructions (in case of multiplications or instructions that use the multiplier) or 4 instructions (for additions/subtractions). The PROM unit can be accessed through an external loading bus, which is the same as the small bus for external communication. In order to distinguish between loading and reading the small bus and the PROM memory, an extra external signal is used alongside the address position. This can be seen as the memory system having a 17-bit address space, instead of 16 bits.

For the Keccak core, we use Bertoni et al.'s VHDL implementation, specifically their *middle-area coprocessor* option [BDPVA12]. We made some adaptations and added an interface for our small bus, in which we can absorb and squeeze data and perform all 24 Keccak rounds. To obtain SHAKE-256 [Nat15] output we use the main unit to prepare the data and only use the Keccak core for the permutation itself.

3.1 Multiplier-accumulator unit: Carmela

We have targeted two distinct architectures; the first relies on a 128-bit MAC (called `Carmela128`), while the second is built upon a 256-bit MAC unit (called `Carmela256`). The design principles are essentially identical, the only difference being the relative size of the units. For that reason, we shall mostly only provide detailed descriptions of the architecture based on `Carmela256` and expect that an understanding of the `Carmela128`-based architecture can be easily inferred, though we highlight the differences wherever necessary. We emphasize that both MAC units were designed with compactness and scalability in mind, but `Carmela128` of course does so more aggressively.

Carmela256. The 256-bit MAC unit features an 8-stage pipeline architecture with 6 stages for the 257-bit signed multiplier and an optimized 546-bit adder. Although the multiplier is constructed with 257 signed bits, it is used as a 256-bit multiplier for both the signed and unsigned cases. Therefore, one of the bits is not fully used in the signed case in exchange for a single bus of 256 bits and a single 256-bit memory word. Indicating whether the input value is signed or not is done by the state machine controlling `Carmela256` directly. The adder, on the other hand, is 546-bit since it is $257 \times 2 + 32$. These extra 32 bits for the accumulator are necessary to perform successive accumulation of several multiplications. While 32 bits are more than strictly necessary to fit the biggest multiplication algorithm of 1024 bits, it would also be enough for potential different but similar algorithms (as the arithmetic operations of SIKE could still be developed further, possibly requiring more additions).

The 257-bit signed multiplier is constructed from small multipliers with the non-standard tiling technique from Roy et al. [RMIT14]. The tiling technique exploits the rectangular shape of the multipliers on more modern FPGA models, which are 25×18-bit signed multipliers. This is done by visualizing the multiplication as a big square, or even a rectangle, and then trying to fill the shape with the minimum amount of rectangles, and being allowed to rotate the rectangles by 90 degrees. In our case we need a 257-bit signed multiplier, which can be done by extending the 256-bit unsigned multiplier of Roy et al., with an additional one bit multiplier to solve the sign. This results in the tiling shown in Figure 5, which has the two extra 1 bit multipliers (though it can be difficult to notice, since the figure has the proportional size).

After multiplication and generation of all partial products, it is necessary to sum and compress into a single value. For this task we employ a 30:5 compressor, followed by a 5:3 compressor, then a 3:2 compressor, and finished with an adder. The entire compression tree of 30:1 has intermediate registers plus 2 extra registers at the end, as shown in Figure 6.

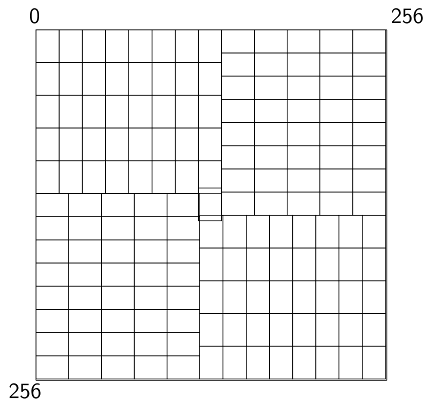

Figure 5: The 257-bit signed multiplier, based on the 256-bit multiplier from [RMIT14].

Those last registers are believed to be rebalanced by the FPGA synthesis tool, since this is not the bottleneck of the maximum period of the circuit.

The final adder in the multiplication was done with a special adder, instead of the carry propagation adder inferred by the synthesis tool. This special adder architecture is given in Figure 7 and the architecture was originally proposed by Nguyen et al. [NPP11]. The adder is a carry-select architecture, where both options for the carry are computed, after which the carry is solved like in a carry lookahead architecture. At the end, with the final carry for each bit the sum output is then generated. This architecture has the lowest latency in Nguyen et al. [NPP11] and the results are very close to those from other architectures as reviewed by Preußer and Krause [PK16].

We show the final architecture with the 257-bit signed multiplier and 546-bit accumulator in Figure 8. The "optimized adder" can add the multiplier output, the multiplication output multiplied by 2 (just a left shift by 1) or the compressor output. The output can then be fed back into the accumulator without change or shifted to the right by the multiplier word size, which is needed for multi-word multiplication. The compressor is combined with the "optimized adder" in order to create addition/subtraction operation with 3 operands, $out = acc + (\mathrm{s} \cdot \mathrm{b}) \pm \mathrm{a}$ or $out = acc - (\mathrm{s} \cdot \mathrm{b}) + \mathrm{a}$. The register "s" is used as a mask for register "b" and to create operations that may or may not add the value of register "b". Finally, the "optimized adder" uses the same AAM architecture discussed before in Figure 7.

The pipeline while doing additions/subtraction operates with 4 stages, because it does not need to wait for the multiplication stages to finish. This 4-stage mode is achieved by the multiplexer added onto the compressor output and the final addition. The register "s" is also only used during addition/subtraction, and therefore only needs 4 register stages to

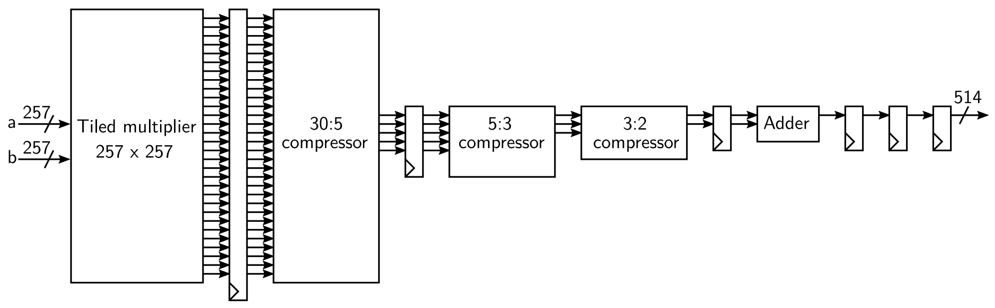

Figure 6: The 257-bit signed multiplier with compressor.

remain synchronized with the pipeline unit.

Carmela128. The biggest difference between the two MACs is the construction of the multiplier itself. `Carmela128` features an 8-stage pipeline architecture with 6 stages for the 129-bit signed multiplier and a 290-bit optimized adder. Similar to `Carmela256`, the multiplier uses 128 bits for both the signed and unsigned cases, while words consist of 16 bits. In contrast to the more complex hand-optimized 257-bit multiplier, the 129-bit signed multiplier with six pipeline stages was constructed by the FPGA synthesis tool. Although we expect this will need more DSPs than what we could achieve by hand, this architecture can easily be deployed in older FPGAs (which do not have rectangular DSPs) like the Xilinx Spartan 6 family. Because it has the same architecture as `Carmela256`, Figure 3 can also be used for `Carmela128`, only changing the sizes of the components.

3.1.1 From a MAC to a finite field unit for up to 1016-bit primes

In this section we focus on `Carmela256` as everything follows completely analogously for `Carmela128`. With the exception of multiplications with/without accumulations and additions/subtractions of 256-bit signed/unsigned values, the MAC cannot perform $\mathbb{F}_p$ operations directly. To construct the operations on top of `Carmela` there are two options: create a state machine to control the hardware pipeline and provide $\mathbb{F}_p$ arithmetic at a higher level, or provide a pipeline in a higher level of `Carmela` and implement the operations in $\mathbb{F}_p$ in software. While both options seem very similar, they can in fact lead to very different constructions. In the case of a software approach, it is necessary for the software to be synchronized with the pipeline or to have a memory that can load/store the 546-bit accumulator values. However, this high bandwidth memory will require a lot of resources in the design so, as a result, we opted for the state machine.

The state machine can use the internal pipeline accumulator registers to synchronize the instructions, while offering the $\mathbb{F}_p$ arithmetic. However, in this case the software cannot be optimized in terms of 257-bit multiplications, but only in terms of $\mathbb{F}_p$ operations. Moreover, to synchronize the operations with the pipeline, the state machine has to offer the same number of operations as the pipeline size, as seen in Figure 9. Therefore, 8 $\mathbb{F}_p$ multiplications have to be performed in parallel, since `Carmela` operates in a mode with an 8-stage pipeline. On the other hand, $\mathbb{F}_p$ addition/subtractions occur as 4 operations in parallel, because `Carmela` operates in 4 pipeline stages. It is of course possible in both cases to perform fewer operations, but the time it will take is the same as if they were 8/4

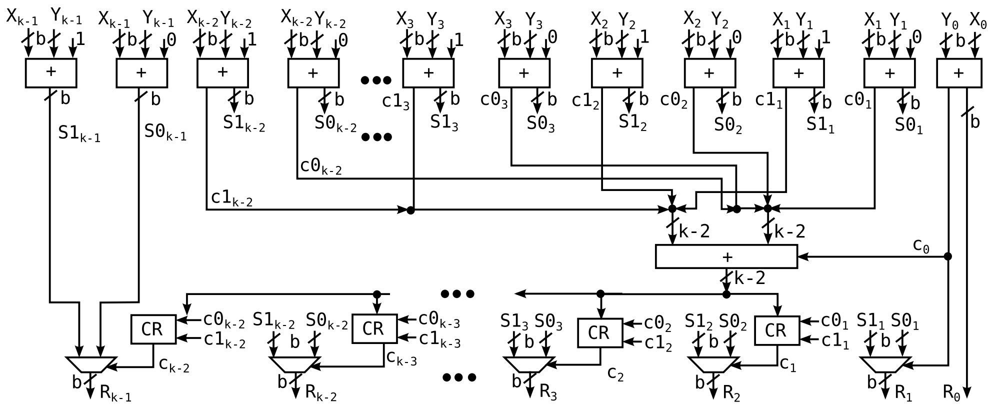

Figure 7: The optimized wide adder architecture Add-Add-Multiplex (AAM) from Nguyen et al. [NPP11]. In our implementation b = 2.

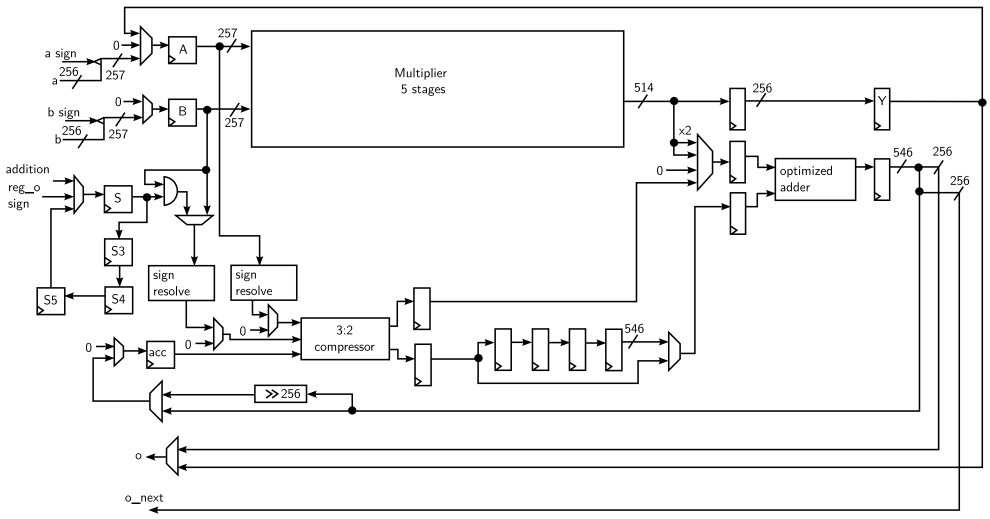

Figure 8: The 8-stage pipeline multiplier accumulator `Carmela256`.

operations, since all parallel operations happen, but only the ones needed have the output enabled.

To keep track of the operands and their addresses, an address resolution unit was added together with the state machine and the `Carmela` pipeline in Figure 10. The address resolution works by receiving an external base address that will be joined by the state machine address to find the final address. The MAC RAM consists of 1024 words of 256 bits (2048 words of 128 bits for `Carmela128`), but for `Carmela` it is subdivided logically as 256 operands of 1024 bits. Thus, it is cheaper to resolve the final address by just appending two (three) bits into the least significant part of the base address. The base address generator also has to behave as a 8/4-stage pipeline, since the base address has to change together with their respective operands.

The state machine implements the operations addition/subtraction without reduction, multiplication without reduction, iterative modular reduction, Montgomery multiplication for any prime p up to 1016 bits, Montgomery multiplication for so-called "Montgomery-friendly" primes p for which $p' = -p^{-1} \bmod R$ is 1, and an optimized squaring variant for each of the multiplications. The addition/subtraction with no reduction instruction is straightforward. The iterative modular reduction is used whenever a public value is between $[-2p, 2p]$, in which case the procedure corrects to a value in the interval $[0, p-1]$. This procedure is only applied to correct data in the very last processing stage.

The multiplication without reduction applies the product scanning method. In this method, the multiplication is done by scanning the product output instead of the operands. This is usually more expensive in terms of loop control logic, but in our case the loops are unrolled. In the case of the Montgomery multiplication for Montgomery-friendly primes, we apply the optimization technique presented in Costello et al. [CLN16], which exploits

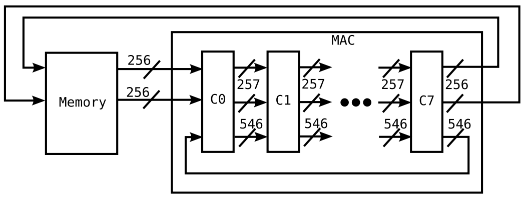

Figure 9: Pipeline with feedback to achieve $\mathbb{F}_p$ operations.

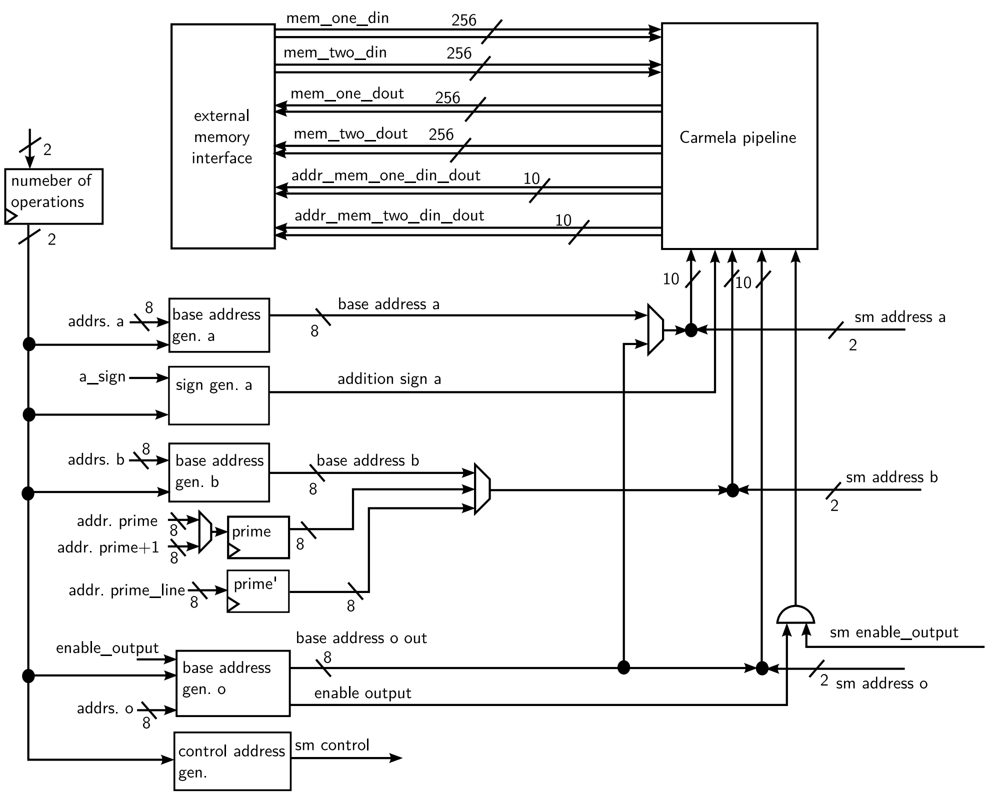

Figure 10: `Carmela256` with address resolution and memory interfaces.

the special form of SIDH/SIKE primes and save several multiplications by computing the radix-2^w Montgomery computation $(a + (ap' \bmod 2^w) \cdot p)/2^w$ as $(a + (a \bmod 2^w) \cdot \hat{p})/2^w$, when $p' = -p^{-1} \bmod 2^w = 1$ for a given word-size w, where a is the input and $\hat{p} = 2^{e_2}3^{e_3}$. In our implementations, we include the optimization as a separate procedure in the state machine, and only apply it to the SIDH/SIKE primes for which $p' = -p^{-1} \bmod 2^w = 1$. Concretely, we apply it to all the cases with exception of `p434` and `p503` on `Carmela256` (i.e., when using $w = 256$). For both cases of the Montgomery multiplication, i.e., with and without exploiting the special prime form, we apply a product-scanning method called Finely Integrated Product Scanning (FIPS) by Koç et al. [KAKJ96].

In addition, the multiplication method that we apply is for signed numbers; not for unsigned numbers, as usually seen in the literature. Montgomery multiplication with signed numbers has been applied before in order to accelerate the multiplication process for Radix-4 multipliers [HW00, TT03]. However, in our case by choosing signed numbers instead of unsigned numbers, it is possible to avoid adding multiples of the prime during the subtractions [Sco17]. Therefore, just as it is possible to use additions with no reduction, the same happens with subtraction. In our case it was opted to apply the Montgomery algorithm with more than 8 extra bits for all 4 primes, but in case it is necessary this value can be increased.

The squaring operations are similar to the respective multiplication operations, except that they are optimized for the same operand input. This optimization makes squares faster than multiplications, yet this operation did not end up being used in the $\mathbb{F}_{p^2}$ operations. This has to do with how the squaring in $\mathbb{F}_{p^2}$ works, which will be explained in Section 3.1.2 in more detail.

Because the state machine has many states (more than 300) we opted to design the state machine as well, and not to use the tool's generator. Figure 11 shows the state

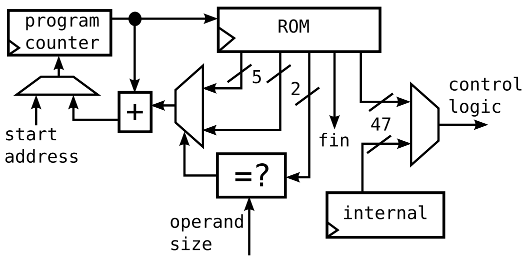

Figure 11: Optimized state machine to operate `Carmela256`.

machine as a ROM which stores all the instructions and a simple control mechanism. The control mechanism is an adder that adds one to go for the next instruction, or adds a bigger value in case a jump might be chosen. The jumps are usually when the algorithm takes a different path, which in our case only happens because of the operand size. The algorithms for operand sizes of 1, 2, 3 and 4 words share some parts, so instead of implementing the 4 algorithms separately we opted to optimize the number of states by trying to share some instructions. The "fin" signal indicates when the last instruction has been executed, and the state machine is ready for the next $\mathbb{F}_p$ operation.

The 129-bit multiplier for `Carmela128` has the same state machine strategy, it just operates with operands of size up to 8, thus it has a lot more states than the 257-bit multiplier. Because our strategy compacts the states into a ROM, the increase of resources is negligible.

3.1.2 Extension field operations

As $p \equiv 3 \bmod 4$, we have an isomorphism $\mathbb{F}_{p^2} \cong \mathbb{F}_p(i)$ where $i^2 = -1$. Fixing the basis $\{1, i\}$ of $\mathbb{F}_{p^2}$ as a 2-dimensional vector space over $\mathbb{F}_p$, we can represent any element $a = a_0 + a_1 \cdot i \in \mathbb{F}_{p^2}$ by the corresponding vector (a_0, a_1). In that case, for elements $a = (a_0, a_1)$ and $b = (b_0, b_1)$ in $\mathbb{F}_{p^2}$, the field operations are given by

$$\begin{aligned}
a + b &= (a_0 + b_0, a_1 + b_1)\,, \\
a - b &= (a_0 - b_0, a_1 - b_1)\,, \\
a \cdot b &= (a_0 \cdot b_0 - a_1 \cdot b_1, a_0 \cdot b_1 + a_1 \cdot b_0)\,, \\
a^2 &= ((a_0 + a_1) \cdot (a_0 - a_1), 2 \cdot a_0 \cdot a_1)\,, \\
a^{-1} &= (a_0 \cdot (a_0^2 + a_1^2)^{-1}, -a_1 \cdot (a_0^2 + a_1^2)^{-1})\,,
\end{aligned}$$

where the operations on the right-hand side are in $\mathbb{F}_p$. We compute the operations in $\mathbb{F}_{p^2}$ directly using the equations above. That is, writing $\mathbf{m}$, $\mathbf{s}$, $\mathbf{a}$ and $\mathbf{i}$ for the cost of a multiplication, squaring, addition/subtraction and inversion in $\mathbb{F}_p$, respectively, and $\mathbf{M}$, $\mathbf{S}$, $\mathbf{A}$ and $\mathbf{I}$ for the cost of a multiplication, squaring, addition/subtraction and inversion in $\mathbb{F}_{p^2}$, respectively, we have $\mathbf{M} = 4\mathbf{m} + 2\mathbf{a}$, $\mathbf{S} = 2\mathbf{m} + 3\mathbf{a}$, $\mathbf{A} = 2\mathbf{a}$ and $\mathbf{I} = 2\mathbf{m} + 2\mathbf{s} + 2\mathbf{a} + \mathbf{i}$. Inversions in $\mathbb{F}_p$ are computed via Fermat's Little Theorem using that $t^{-1} = t^{p-2}$ for any (non-zero) $t \in \mathbb{F}_p$. Note that we use projective coordinates for all elliptic-curve operations, making the efficiency of the inversion operation of limited interest. We therefore favor Fermat inversion over more complex methods such as the constant-time gcd-based modular inversion of Bernstein and Yang [BY19].

Typically, since multiplications are the most expensive operation in $\mathbb{F}_p$, implementations perform multiplication in $\mathbb{F}_{p^2}$ at a cost of $3\mathbf{m} + 5\mathbf{a}$ by using the Karatsuba method [KO62]. However, this trade-off comes at the expense of more dependent operations (see Figure 12). As depicted in Figure 12, the schoolbook method can be performed with a latency of $1\mathbf{m} + 1\mathbf{a}$, while the Karatsuba method has a latency of $1\mathbf{m} + 3\mathbf{a}$. Although the extra

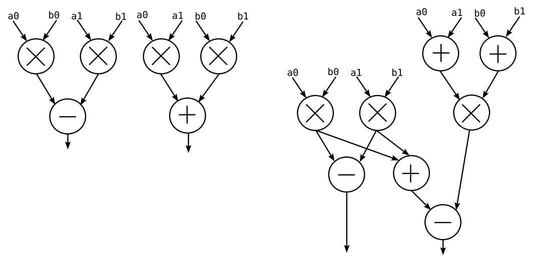

Figure 12: $\mathbb{F}_{p^2}$-multiplication – schoolbook (left), Karatsuba (right).

additions/subtractions do not contribute significantly to the total time of the multiplication, the overhead is not negligible either. More importantly, the multiplier unit `Carmela` is designed to perform 8 $\mathbb{F}_p$ multiplications in parallel, fitting exactly 2 $\mathbb{F}_{p^2}$ operations if the schoolbook method is adopted.

The square operation in $\mathbb{F}_{p^2}$ is also not performed with the Karatsuba optimization, nor schoolbook, but simply as a $\mathbb{F}_{p^2}$ multiplication. This is because the $\mathbb{F}_{p^2}$ squaring needs to perform at least one regular multiplication (i.e., $a_0 \cdot a_1$). Because the multiplier always performs 8 $\mathbb{F}_p$ multiplications in parallel, the cost of squaring by using the square operation will be higher than using the regular multiplication directly. For this reason, we do not use the optimized $\mathbb{F}_{p^2}$ squaring in this case.

In terms of improvements, there are several possibilities and trade-offs. One possibility is to implement a pipeline aimed for the Karatsuba method. In this case, one would need 6 stages if aiming at two $\mathbb{F}_{p^2}$ operations in parallel, or even 3 stages for one operation at a time. It is also possible to reduce the pipeline to 4 stages and limit it to one operation at a time using the schoolbook algorithm. Note that reducing the number of stages would increase the time each cycle takes, with the control instructions having a bigger contribution in the total time. Another potential modification in our architecture is not to have the multiplier output the multiplication with one value, but instead the carry save notation, and then having only one large adder that is used in conjunction with the accumulator. Other ideas involve the design of a basic 64-bit MAC instead of relatively large 128- and 256-bit MACs, and the direct implementation of the $\mathbb{F}_{p^2}$ arithmetic in the state machine.

3.2 Instruction set and memory arrangement

The template for the instructions of the main processor is depicted in Figure 13. The main processor instructions should have the two most significant bits (63 and 62) be zero, otherwise it is a coprocessor instruction. The 61st bit is reserved and should be 0, while the next 6 bits (60 to 55) are used to infer the instruction type. The input operands are identified between bits 0 to 18 and 19 to 37 for "a" and "b", respectively. The output operand "o" is determined by bits 38 to 54. For each input operand there is the "Cx" flag on bits 16 and 35 respectively, which distinguishes the value "mem(a,b)" as a constant or a memory address. The bit "Dir" indicates when the operand value is a memory address that contains the value applied in the operation or if the operand value is a memory address that points to the memory address that has the real value. Finally, the bit "Sign" shows if the operand values are signed or unsigned. Not all bits are active in all instructions, since the SIKE protocol does not need all possible combinations. By not implementing all possible options we reduce the amount of resources in the decoding unit.

Some of the main instructions have different variations according to which operand is going to be used. For example, the push instruction has variants "pushf" and "'pushm". The appended "f" and "m" indicate instructions for 256-bit data (or 128-bit data for `Carmela128`) and operands up to 1024 bits, respectively. The regular push instruction will take a 16 bits value and push into memory. The "f" variant is meant to be applied to the MAC RAM values, that can be addressed in 1024 words of 256 bits (or 2048 words of 128 bits for `Carmela128`). However, most of the time we are not working on the bare 256 or 128-bit words, but on the $\mathbb{F}_p$ operands which can be any 256-bit or 128-bit multiple (up to 1024 bits). For those cases, the preferred instruction is the one that ends with an "m", as it will work directly with operand values. For the full set of instructions of the main unit we refer to Figure 17 in the Appendix.

The instruction template for the coprocessor `Carmela` is shown in Figure 14. The `Carmela` coprocessor instruction is identified as "01" in bits 63 to 62. Because `Carmela` has fewer instructions, only 4 bits are used to identify the type with 2 inputs and 1 output. Just as in the main instruction template, the bit "Dir" is used to distinguish values that hold memory addresses or values that hold a memory address that points to the effective memory address. The "Dir" bit is also used to show if the output is disabled or not by setting it as 1 and "mo" as 0. The bit "Sign" in this case is not used to identify an operation as signed or unsigned, but to distinguish addition and subtraction during the addition/subtraction instruction ($\text{mo} = \text{mb} \pm \text{ma}$). Since addition/subtraction instructions only need 4 operations in parallel, the instruction in this case only uses the first 4 operands (0 to 3). The instruction operand not used for addition/subtraction could be optimized to not have the "Sign" bit, but it was added to the instruction for consistency. The "'Dir" bit can also work with the "RD register 0" least significant bit (LSB) or most significant bit (MSB). In this mode, the "RD register 0" bit replaces the LSB in the instruction memory value. This makes it possible to change which pointer will be used depending of the "RD register 0" bit.

In order to be able to perform all 8/4 operations in parallel `Carmela` instructions must be loaded serially. Therefore, all 8/4 operations are loaded one after another. In case one wants to perform fewer operations, it is still necessary to load all 8/4 operations, but the ones not used should have the output disabled.

The internal small bus can access all SIKE components through a 16-bit address region, which is detailed in Figure 15. The first positions belong to the MAC RAM, which includes some reserved space in case it is necessary to increase its size. The program part holds the SIKE processor instructions. The ALU RAM is the area used for control and to perform the isogeny tree algorithm of SIDH. The Keccak core memory positions are to send/receive data and commands to the core. The status register only shows if the SIKE processor is free or not. Operands size is for the size of the operands primes (0 = up to 256, 1 = up to 512, 2 = up to 768, 3 = up to 1024, and in the case of `Carmela128` the values go from 0 to 7 up to 1024). The flag is another register that is used to help debug the processor, which can be also seen as the external communication output of the processor.

The program counter register has two purposes. The first is to keep track of the current instruction address, and the second purpose is to start any procedure inside the SIKE processor. For example, if the desired function to run is in position 3 of the program memory, we just write in the program counter register the value 3 and it will start the execution. The execution continues until a last instruction called "fin" is executed, then the processor stops and waits until a new value is written on the program counter. For the full set of instructions of the coprocessor we refer to Figure 18 in the Appendix.

main flag	append	main internal	operand o		operand b				operand a			
63 62 00	61 0	60 ... 55 type	54 Dir	53 ... 38 memo	37 Sign	36 Dir	35 Cx	34 ... 19 memb	18 Sign	17 Dir	16 Cx	15 ... 0 mema

Figure 13: Main instruction template for the SIKE processor.

carmela flag	append	carmela internal	operand o		operand b				operand a			
63 62 01	61 ... 59 000	58 ... 55 type	54 Dir	53 ... 38 Mo	37 0	36 Dir	35 En	34 ... 19 Mb	18 Sign	17 Dir	16 En	15 ... 0 Ma

Figure 14: Instruction template for the coprocessor `Carmela`.

3.3 Software elliptic curve and isogeny computations

As discussed in Section 3, elliptic curve and isogeny computations, including the three-point differential ladder and large-degree isogeny computation, are done in software; see Figure 2. Our software implementation is very similar to the C implementation of SIKE in the SIDH library [CLN18], since we use the same structure of function calls to implement key generation, encapsulation and decapsulation. The only difference is that we optimize the isogeny computation, three-point differential ladder and field inversion to perform two $\mathbb{F}_{p^2}$ operations in parallel whenever possible. Only in very few instances do we have to perform one $\mathbb{F}_{p^2}$ operation at a time.

Because the entire underlying arithmetic described in Section 3.1 executes in a constant amount of clock cycles, and the software uses constant-time routines exactly like in the C implementation of the SIDH library, our implementation is also constant-time. The only source of timing variability is the parameter size, since the arithmetic unit operates in multiples of 256 for `Carmela256` and 128 for `Carmela128`. However, these sizes are public and do not leak sensitive information. For the three-point differential ladder (the only function that directly handles the secret key), we always assume the maximum number of bits for the expected secret. Some timing variable components like caches are not present, and memory accesses take the same amount of time irregardless of the position. Finally, the main CPU itself is also constant time, since all instructions take the same amount of time and every instruction waits for the previous one to finish before starting.

Although the design is constant-time, it is not safe against differential side-channel attacks like DPA. In those cases it is necessary to apply countermeasures such as scalar and projective coordinate randomization to the ladder steps. These countermeasures can be implemented in software together with a random number generator in the architecture. It is worthy to mention that some previous works already implemented such countermeasures [KAK+19].

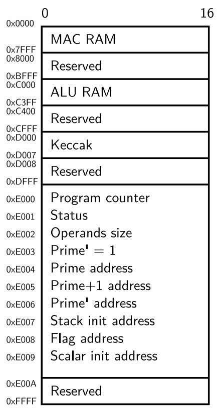

Figure 15: Memory arrangement of the SIKE processor.

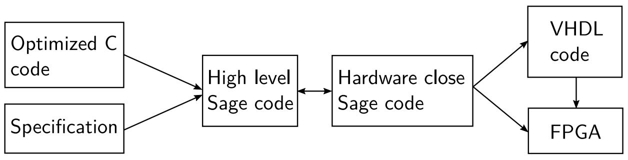

Figure 16: How the testing and coding was done for VHDL.

4 Results and Analysis

The entire hardware architecture was described and tested on VHDL. To test the implementations we used the procedure depicted in Figure 16. The SIKE protocol was first implemented in Python with the Sage environment [The17] for testing purposes. Then a second Sage code that mimics the procedures used by the hardware, like the Montgomery multiplication, was done and then tested against the higher level Sage code. Finally, the same tests applied to test the second Sage code itself were used to test the VHDL code. When the VHDL was simulated, the code was synthesized in the Zynq platform and then the same tests used for the VHDL were applied. All tests were done in parts, testing each function separately. This approach eased the debugging process of the architecture.

The post-place and route results that we obtained for our architectures with Xilinx ISE 14.7 (Virtex 7 and Artix 7) and Vivado 2019.1 (Zynq 7020) are displayed in Table 2. For our tests on the Zynq platform, we used a Zedboard development kit [Avn19]. For Artix 7, which is the platform recommended by NIST in the ongoing PQC standardization process [Moo19], we used an Arty A7 development kit [Dig19] that contains the Artix 7 FPGA model XC7A100TCSG324-3. For Virtex 7, we targeted the FPGA model XC7V690T, which was chosen to ease comparisons with previous work by Koziel et al. [KAMK16, KAMK18, KAK$^+$19] and Roy and Mukhopadhyay [RM19].

Table 2 nicely illustrates the effect of using each of our two `Carmela` options. As expected, the 128-bit architecture consumes significantly fewer slices and DSPs, reflecting the fact that the `Carmela` unit needs less logic, multipliers and registers. It also requires fewer BRAMs, due to the reduced bandwidth: to retrieve one 128-bit word we need 4 BRAMs of 32 bits, while for 256 bits we need twice that number. In the case of the operating frequency, the `Carmela128`-based architecture only achieves a slightly higher frequency, given that the entire control mechanism remains expensive even though the `Carmela` unit is smaller.

Table 2: Resources required for the `Carmela128` and `Carmela256`-based architectures on several platforms. The max period is measured in nanoseconds (ns).

Mul.	FPGA	Slices	LUTs	FFs	BRAMs	DSPs	Max period
128	Virtex 7 690T	3 415	10 937	7 132	21	57	6.570
	Artix 7 100T	3 512	10 976	7 115	21	57	6.993
256	Virtex 7 690T	7 408	21 210	13 657	38	162	7.034
	Artix 7 100T	7 491	22 595	11 558	37	162	9.181

Timing results for $\mathbb{F}_p$ operations. We begin by presenting the intermediate cycle counts for the $\mathbb{F}_p$ operations performed by `Carmela128` and `Carmela256` (i.e., multiplication, squaring and addition/subtraction, see Table 3). Since we aimed at always performing 2 $\mathbb{F}_{p^2}$ operations in parallel, the multiplication and squaring operations execute 8 $\mathbb{F}_p$ multiplications resp. squarings in parallel, while addition/subtraction and iterative reduction perform 4 operations in parallel. We refer to Section 3.1.2 for more details.

Table 3: Cycle counts of $\mathbb{F}_p$ operations for the `Carmela128` and `Carmela256`-based architectures. The top column shows the maximum bit size of the operands. The operations that use the multiplier: multiplication (Mult.) and squaring (Square) with/without modular reduction (red.) perform 8 operations in parallel. The cycles are measured after all 8 outputs have been written to memory. The addition/subtraction (Add/Sub) and iterative reduction only operate on 4 inputs.

Mul.	**Operation**	**126**	**254**	**382**	**510**	**638**	**766**	**894**	**1022**
128	Mult. no red.	18	42	82	138	210	298	402	522
	Mult. with red.	42	90	178	298	450	634	850	1098
	Mult. with red. $p' = 1$	–	58	130	234	330	490	626	842
	Square no red.	18	34	58	90	130	178	234	298
	Square with red.	42	82	154	250	370	514	682	874
	Square with red. $p' = 1$	–	50	106	186	250	370	458	618
	Add/Sub no red.	10	14	18	22	26	30	34	38
	Iterative red.	22	34	46	58	70	82	94	106
256	Mult. no red.	18	18	42	42	82	82	138	138
	Mult. with red.	42	42	90	90	178	178	298	298
	Mult. with red. $p' = 1$	–	–	58	58	130	130	234	234
	Square no red.	18	18	34	34	58	58	90	90
	Square with red.	42	42	82	82	154	154	250	250
	Square with red. $p' = 1$	–	–	50	50	106	106	186	186
	Add/Sub no red.	10	10	14	14	18	18	22	22
	Iterative red.	22	22	34	34	46	46	58	58

As `Carmela128` and `Carmela256` support arbitrary primes (i.e., not only SIKE primes), we can compare the results of the multiplication routine with existing ECC literature. For example, Järvinen et al. [JMAL16] construct a 127-bit solution consisting of a 64-bit pipelined multiplier accumulator that has 7 stages, for support of the Four$\mathbb{Q}$ [CL15] curve defined over a quadratic extension of $\mathbb{F}_{2^{127}-1}$. As such, it supports up to 7 multiplications in parallel and achieves a latency of 20 cycles per multiplication with 16 DSPs. For comparison, `Carmela128` requires 42 cycles for 8 multiplications (5.25 cycles per multiplication) with 57 DSPs. We achieve similar speed/area, where we lean more towards area to be able to support larger primes at low latency as well. The 256-bit version `Carmela256` also takes 42 for cycles for 8 multiplications with 162 DSPs, though it is clearly too large for the intended prime size.

Similarly, Sasdrich and Güneysu [SG17] aim for a specific 448-bit prime to support Curve448 [LHT16]. They achieve 36 cycles per multiplication with 33 DSPs. In our case, `Carmela128` takes 298 cycles for 8 multiplications (37.25 cycles per multiplication) at a cost of 57 DSPs, while `Carmela256` requires 90 cycles for 8 multiplications (11.25 cycles per multiplication) at the cost of 162 DSPs. As was the case for the work of Järvinen et al. [JMAL16], we note that the modular reductions from Sasdrich and Güneysu significantly benefit from the simple prime structure, which we do not exploit so as to be able to support all primes up to 1022 bits.

Comparison with other SIDH/SIKE implementations. We compare our SIKE results with other results in the literature in Table 4. We focus on the Virtex 7 690T platform because, to our knowledge, it has been the only target of previous SIDH and SIKE implementations. As advertised, existing works [KAMK16, KAMK18, ACC+19b, RM19, KAK+19] achieve high-speed at the expense of a much higher amount of resources. For example, in comparison with our 256-bit architecture, the most recent SIKE implementation

Table 4: Comparison of SIKE implementations on the Xilinx Virtex 7 690T. The frequency ("Freq.") is measured in Megahertz (MHz) and the total time (`Encaps` + `Decaps` for SIKE, a full ephemeral key exchange for SIDH) in milliseconds (ms). The ST resp. DT column displays the number of slices and DSPs respectively times the time divided by 1000, rounded to the nearest multiple of 1000 resp. 100. Our architecture works directly for any parameter set `pXXX` for `XXX` $\in \{434, 503, 610, 751\}$ and does not need to be reprogrammed. Works emphasized with a † symbol only implement SIDH, while the others do SIKE.

Reference	Par.	Slices	DSP	BRAM	Freq.	Time	ST	DT
[KAMK16]†	`p503`	8 918	192	40.0	181.4	20.9	186	4.0
	`p751`	11 801	282	47.0	177.3	46.3	546	13.1
[KAMK18]†	`p503`	7 491	192	43.5	202.1	16.5	124	3.2
	`p751`	11 277	288	60.5	204.9	36.4	410	10.5
[RM19]†	`p751`	18 711	294	22.5	225.7	31.6	591	9.3
[ACC+19b]	`p751`	16 756	376	56.5	198.0	33.4	560	12.6
[KAK+19]	`p503`	9 514	264	34.0	171.2	13.6	129	3.6
	`p751`	17 530	512	43.5	167.4	26.9	472	13.8
Ours (128)	`pXXX`	3 415	57	21.0	152.2	50.4	172	2.9 (`p434`)
						59.5	203	3.4 (`p503`)
						107.2	366	6.1 (`p610`)
						179.6	613	10.2 (`p751`)
Ours (256)	`pXXX`	7 408	162	38.0	142.2	24.3	180	3.9 (`p434`)
						28.7	212	4.6 (`p503`)
						51.8	384	8.4 (`p610`)
						60.8	450	9.8 (`p751`)

by [KAK+19] is 53% and 56% faster for `SIKEp503` and `SIKEp751`, respectively. However, for the largest parameter set (i.e., `SIKEp751`) our architecture requires up to 58% fewer slices, 13% fewer BRAMs and 68% fewer DSPs. We remark that DSPs are significantly expensive hardware resources. Moreover, our architecture supports all four different SIKE parameters sets (and any other primes if necessary), while that of [KAK+19] only supports `SIKEp751` unless the device is reprogrammed. For our 128-bit architecture, the implementation of [KAK+19] is 77% and 85% faster for `SIKEp503` and `SIKEp751` respectively, but we need 81% fewer slices, 52% fewer BRAMs and 89% fewer DSPs to support `SIKEp751` (and other parameter sets).

Since we present a speed/area trade-off, we also include a proper metric to compare against existing literature. As previous work used the product of slices and time (ST) as a measure, we include this as well. In this metric, `Carmela256` has fairly similar performance to `Carmela128`, but scales much better to larger primes, as can be seen from the results for `SIKEp751`. In comparison to the best recent work on SIKE [KAK+19] we do worse for `SIKEp503`, but can see that `Carmela256` achieves a more favorable trade-off than [KAK+19] for `SIKEp751`. However, we note that this metric excludes the expensive DSP resources. For that purpose, we also include the product of DSPs and time (DT) as a metric. In that case `Carmela128` performs better than `Carmela256` for the smaller SIKE primes, while `Carmela256` is superior for `SIKEp751`. We also observe that `Carmela128` achieves better results than the work of Koziel et al. [KAK+19], while `Carmela256` outperforms it for `SIKEp751`. The work of Roy and Muhopadhyay [RM19] has even better results than `Carmela256` for `SIKEp751`, though it is worse in the ST metric. We emphasize that neither of the metrics provides a complete picture; the relevant trade-off depends on the context

Table 5: Comparison of PQC implementations on the Xilinx Artix 7 100T. The frequency ("Freq.") is measured in Megahertz (MHz), the total time (`Encaps` + `Decaps` for BIKE, FrodoKEM and SIKE, a full ephemeral key exchange for NewHope) in milliseconds (ms). Our architecture works directly for any parameter set `pXXX` for `XXX` $\in \{434, 503, 610, 751\}$ and does not need to be regenerated.

Reference	Algorithm	Slices	DSPs	BRAMs	Freq.	Time
[OG19]	NewHope-Simple (Server/Client)	1 708 1 483	2 2	4 4	125.0 117.0	2.9
[ABB+19]	BIKE	1 559	-	13	161.3	10.2
[HOKG18]	FrodoKEM-640 (Encaps/Decaps)	1 855 1 992	1 1	11 16	167.0 162.0	40.0
Ours (128)	**SIKE**	3 512	57	21	143.0	53.6 (`p434`) 63.4 (`p503`) 114.1 (`p610`) 191.2 (`p751`)
Ours (256)	**SIKE**	7 491	162	37	108.9	31.7 (`p434`) 37.4 (`p503`) 67.6 (`p610`) 79.3 (`p751`)

in which the architecture is used and the relative cost of the different components.

A similar trade-off can be observed against works only implementing the SIDH protocol, such as [KAMK18] and [RM19]. In this case, however, one needs to take into account some additional costs to support SIKE, including the use of the Keccak core to implement SHAKE-256.

We note that [KAK+19] includes two countermeasures against differential power analysis, namely, scalar randomization and projective coordinate randomization. These techniques, which are applied during key generation and decapsulation, exhibit a very small overhead and are easy to implement in the proposed architectures using our hardware/software co-design. As stated before, unlike previous results our architectures can work on all four currently available primes (and any other future prime as long as it is smaller than 1016 bits) without having to be re-synthesized. For any new prime, only the relevant constants and public parameters need to be uploaded.

Comparison with other PQC implementations. We compare our SIKE results with other representative post-quantum key-exchange algorithms in the literature in Table 5. We focus on the Artix 7 100T platform, which is widely used and has been recommended by NIST in the ongoing PQC standardization process.

As can be seen, SIKE demands more resources and is computationally more expensive than competing alternatives. However, it should be noted that the proposed architectures advance the state-of-the-art and reduce that gap significantly, arguably demonstrating that SIKE, which features the smallest public keys in the NIST PQC process, can be efficiently implemented for embedded applications.

5 Conclusions and Future Work

We presented the first hardware/software co-design for the SIKE scheme, and the first hardware implementation of the new round 2 parameters `SIKEp434` and `SIKEp610` (in addition to SIKEp503 and SIKEp751). In contrast to earlier works, we focus on compactness and scalability instead of raw speed. We believe this presents interesting new trade-offs; the 128-bit architecture is roughly 80% smaller and slower, while the 256-bit design is about 55% smaller and slower compared to existing implementations. Besides being more compact, the co-design approach leads to attractive scalability properties. For any prime p of at most 1016 bits we support $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ arithmetic, and any SIDH and SIKE implementation built on top of it. Therefore, instead of regenerating the circuit to change parameter sets we can simply upload the respective SIKE constants and public parameters. In particular, this makes it very easy to update the parameters sets defined in Round 2 of SIKE's NIST submission if they are modified for Round 3.

Moreover, the proposed coprocessor can also be used for performing the prime field arithmetic used in classical elliptic-curve cryptography. For example, adding a single additional routine in software can very cheaply lead to support of discrete-log-based protocols via the Montgomery curve "BigMont" proposed in [CLN16] or any of the classical prime-field curves standardized by NIST [Nat13]. Combining this with SIKE gives rise to a hybrid protocol that may be of serious interest while post-quantum protocols are still being analyzed and standardized. Though out of scope for this work, we consider this an interesting research direction.

Another interesting line of research would be to include more extensive side-channel countermeasures. Although our implementation is constant-time and, as such, protected against simple side-channel and timing attacks, more sophisticated countermeasures would include those that protect against more advanced attacks. We also consider these to be out of scope for this work, but believe the hardware/software co-design makes some obvious candidates (e.g., projective coordinate and scalar randomization) straightforward to adopt.

Finally, the instruction set was designed to provide the operations needed for the SIDH/SIKE functions, but should also work with other programs or procedures. Future work could involve the change of the instruction set to a well-known and widely adopted ISA, such as RISC-V.

Acknowledgments

We would like to thank the TCHES reviewers for their useful comments and feedback. We also thank Xilinx for the University license used during the synthesis and simulation of our architecture. Lejla Batina, Pedro Maat C. Massolino and Joost Renes were supported by the Technology Foundation STW (project 13499 - TYPHOON), from the Dutch government.

References

[ABB+19] N. Aragon, P. S. L. M. Barreto, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, S. Gueron, T. Güneysu, C. A. Melchor, R. Misoczki, E. Persichetti, N. Sendrier, J.-P. Tillich, V. Vasseur, , and G. Zémor. Bit Flipping Key Encapsulation – Submission to Round 2 of NIST's Post-Quantum Cryptography Standardization Process, 2019. Available at https://bikesuite.org.

[ACC+17] Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Amir Jalali, David Jao, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Joost Renes, Vladimir Soukharev, and David Urbanik. Supersingular Isogeny Key Encapsulation – Submission to the NIST's Post-Quantum Cryptography Standardization Process, 2017. Available at https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-1/submissions/SIKE.zip.

[ACC+19a] Gora Adj, Daniel Cervantes-Vázquez, Jesús-Javier Chi-Domínguez, Alfred Menezes, and Francisco Rodríguez-Henríquez. On the Cost of Computing Isogenies Between Supersingular Elliptic Curves. In Carlos Cid and Michael J. Jacobson Jr., editors, *Selected Areas in Cryptography - SAC 2018*, volume 11349 of *Lecture Notes in Computer Science*, pages 322–343. Springer, 2019.

[ACC+19b] Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Amir Jalali, David Jao, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Geovandro Pereira, Joost Renes, Vladimir Soukharev, and David Urbanik. Supersingular Isogeny Key Encapsulation – Submission to Round 2 of NIST's Post-Quantum Cryptography Standardization Process, 2019. Available at https://sike.org.

[AFJ14] Reza Azarderakhsh, Dieter Fishbein, and David Jao. Efficient Implementations of A Quantum-Resistant Key-Exchange Protocol on Embedded systems. Technical report, Center for Applied Cryptographic Research (CACR) – University of Waterloo, 2014. Available at http://cacr.uwaterloo.ca/techreports/2014/cacr2014-20.pdf.

[AJK+16] Reza Azarderakhsh, David Jao, Kassem Kalach, Brian Koziel, and Christopher Leonardi. Key Compression for Isogeny-Based Cryptosystems. In Keita Emura, Goichiro Hanaoka, and Rui Zhang, editors, *ACM International Workshop on ASIA Public-Key Cryptography (AsiaPKC 2016)*, pages 1–10. ACM, 2016.

[Avn19] Avnet. Zedboard, 2019. http://zedboard.org/product/zedboard.

[BDPVA12] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak in VHDL, 2012. https://keccak.team/hardware.html.

[BJS14] Jean-François Biasse, David Jao, and Anirudh Sankar. A Quantum Algorithm for Computing Isogenies between Supersingular Elliptic Curves. In Willi Meier and Debdeep Mukhopadhyay, editors, *Progress in Cryptology - INDOCRYPT 2014*, volume 8885 of *Lecture Notes in Computer Science*, pages 428–442. Springer, 2014.

[BY19] Daniel J. Bernstein and Bo-Yin Yang. Fast constant-time gcd computation and modular inversion. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(3):340–398, May 2019.

[CH17] Craig Costello and Hüseyin Hisil. A Simple and Compact Algorithm for SIDH with Arbitrary Degree Isogenies. In *Advances in Cryptology - ASIACRYPT 2017*, volume 10625 of *Lecture Notes in Computer Science*, pages 303–329. Springer, 2017. https://ia.cr/2017/504.

[CJL+17] Craig Costello, David Jao, Patrick Longa, Michael Naehrig, Joost Renes, and David Urbanik. Efficient Compression of SIDH Public Keys. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology - EUROCRYPT 2017*, volume 10210 of *Lecture Notes in Computer Science*, pages 679–706. Springer, 2017.

[CL15] Craig Costello and Patrick Longa. Four$\mathbb{Q}$: Four-Dimensional Decompositions on a $\mathbb{Q}$-curve over the Mersenne Prime. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology – ASIACRYPT 2015*, pages 214–235, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[CLN16] Craig Costello, Patrick Longa, and Michael Naehrig. Efficient Algorithms for Supersingular Isogeny Diffie-Hellman. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology - CRYPTO 2016*, volume 9814 of *Lecture Notes in Computer Science*, pages 572–601. Springer, 2016.

[CLN18] Craig Costello, Patrick Longa, and Michael Naehrig. SIDH Library. https://github.com/Microsoft/PQCrypto-SIDH, 2016–2018.

[CLN$^+$19] Craig Costello, Patrick Longa, Michael Naehrig, Joost Renes, and Fernando Virdia. Improved Classical Cryptanalysis of the Computational Supersingular Isogeny Problem, 2019. Available at https://eprint.iacr.org/2019/298.pdf.

[DG16] Christina Delfs and Steven D. Galbraith. Computing isogenies between supersingular elliptic curves over $\mathbb{F}_p$. *Des. Codes Cryptography*, 78(2):425–440, 2016.

[Dig19] Digilent. Arty A7: Artix-7 FPGA Development Board for Makers and Hobbyists, 2019. https://store.digilentinc.com/arty-a7-artix-7-fpga-development-board-for-makers-and-hobbyists/.

[FJP14] Luca De Feo, David Jao, and Jérôme Plût. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. *J. Mathematical Cryptology*, 8(3):209–247, 2014.

[FO13] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure Integration of Asymmetric and Symmetric Encryption Schemes. *J. Cryptology*, 26(1):80–101, 2013.

[Gal18] Steven D. Galbraith. Authenticated key exchange for SIDH, 2018. Available at https://eprint.iacr.org/2018/266.pdf.

[GPS17] Steven D. Galbraith, Christophe Petit, and Javier Silva. Identification Protocols and Signature Schemes Based on Supersingular Isogeny Problems. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology - ASIACRYPT 2017*, volume 10624 of *Lecture Notes in Computer Science*, pages 3–33. Springer, 2017.

[GPST16] Steven D. Galbraith, Christophe Petit, Barak Shani, and Yan Bo Ti. On the Security of Supersingular Isogeny Cryptosystems. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology - ASIACRYPT 2016*, volume 10031 of *Lecture Notes in Computer Science*, pages 63–91, 2016.

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A Modular Analysis of the Fujisaki-Okamoto Transformation. In Yael Kalai and Leonid Reyzin, editors, *Theory of Cryptography (TCC 2017)*, volume 10677 of *Lecture Notes in Computer Science*, pages 341–371. Springer, 2017.

[HOKG18] James Howe, Tobias Oder, Markus Krausz, and Tim Güneysu. Standard lattice-based key encapsulation on embedded devices. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):372–393, 2018.

[HW00] Jin-Hua Hong and Cheng-Wen Wu. Radix-4 modular multiplication and exponentiation algorithms for the RSA public-key cryptosystem. In *Proceedings 2000. Design Automation Conference. (IEEE Cat. No.00CH37106)*, pages 565–570, June 2000.

[JF11] David Jao and Luca De Feo. Towards Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies. In Bo-Yin Yang, editor, *Post-Quantum Cryptography (PQCrypto 2011)*, volume 7071 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2011.

[JMAL16] Kimmo Järvinen, Andrea Miele, Reza Azarderakhsh, and Patrick Longa. Four$\mathbb{Q}$ on FPGA: New Hardware Speed Records for Elliptic Curve Cryptography over Large Prime Characteristic Fields. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems – CHES 2016*, pages 517–537, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[JS14] David Jao and Vladimir Soukharev. Isogeny-Based Quantum-Resistant Undeniable Signatures. In Michele Mosca, editor, *Post-Quantum Cryptography (PQCrypto 2014)*, volume 8772 of *Lecture Notes in Computer Science*, pages 160–179. Springer, 2014.

[JS19] Samuel Jaques and John M. Schanck. Quantum Cryptanalysis in the RAM Model: Claw-Finding Attacks on SIKE. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology - CRYPTO 2019*, volume 11692 of *Lecture Notes in Computer Science*, pages 32–61. Springer, 2019.

[KAK$^+$19] Brian Koziel, A.-Bon Ackie, Rami El Khatib, Reza Azarderakhsh, and Mehran Mozaffari Kermani. SIKE'd up: Fast and secure hardware architectures for supersingular isogeny key encapsulation, 2019. Available at https://eprint.iacr.org/2019/711.pdf.

[KAKJ96] Çetin K. Koç, Tolga Acar, and Burton S. Kaliski Jr. Analyzing and comparing Montgomery multiplication algorithms. *Micro, IEEE*, 16(3):26–33, Jun 1996.

[KAMK16] Brian Koziel, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. Fast Hardware Architectures for Supersingular Isogeny Diffie-Hellman Key Exchange on FPGA. In Orr Dunkelman and Somitra Kumar Sanadhya, editors, *Progress in Cryptology - INDOCRYPT 2016*, volume 10095 of *Lecture Notes in Computer Science*, pages 191–206. Springer, 2016.

[KAMK18] Brian Koziel, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. A High-Performance and Scalable Hardware Architecture for Isogeny-Based Cryptography. *IEEE Transactions on Computers*, pages 1594–1609, 2018. https://doi.org/10.1109/TC.2018.2815605.

[KJA$^+$16] Brian Koziel, Amir Jalali, Reza Azarderakhsh, David Jao, and Mehran Mozaffari-Kermani. NEON-SIDH: Efficient Implementation of Supersingular Isogeny Diffie-Hellman Key Exchange Protocol on ARM. In Sara Foresti and Giuseppe Persiano, editors, *Cryptology and Network Security (CANS 2016)*, volume 10052 of *Lecture Notes in Computer Science*, pages 88–103. Springer, 2016.

[KO62] A. Karatsuba and Yu. Ofman. Multiplication of Many-Digital Numbers by Automatic Computers. In *Proceedings of the USSR Academy of Sciences*, pages 293–294, 1962.

[LHT16] Adam Langley, Mike Hamburg, and Sean Turner. Elliptic Curves for Security. RFC 7748, RFC Editor, 2016.

[Lon18] Patrick Longa. A note on post-quantum authenticated key exchange from supersingular isogenies, 2018. Available at https://eprint.iacr.org/2018/267.pdf.

[Moo19] Dustin Moody. Round 2 of the NIST PQC "Competition" – What was NIST thinking?, 2019. Presentation PQCrypto 2019, https://csrc.nist.gov/CSRC/media/Presentations/Round-2-of-the-NIST-PQC-Competition-What-was-NIST/images-media/pqcrypto-may2019-moody.pdf.

[Nat13] National Institute for Standards and Technology. Federal Information Processing Standards Publication 186-4. Digital signature standard. Technical report, NIST, 2013.

[Nat15] National Institute for Standards and Technology. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. Technical report, NIST, 2015.

[Nat18a] National Institute of Standards and Technology (NIST). Post-Quantum Cryptography Standardization, 2017–2018. https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization.

[Nat18b] National Institute of Standards and Technology (NIST). Post-Quantum Cryptography Standardization – Round 1 Submissions, 2018. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions.

[NPP11] H. D. Nguyen, B. Pasca, and T. B. Preußer. FPGA-Specific Arithmetic Optimizations of Short-Latency Adders. In *2011 21st International Conference on Field Programmable Logic and Applications*, pages 232–237, Sept 2011. http://dx.doi.org/10.1109/FPL.2011.49.

[NR19] Michael Naehrig and Joost Renes. Dual Isogenies and Their Application to Public-key Compression for Isogeny-based Cryptography. In *Advances in Cryptology - ASIACRYPT 2019 (to appear)*, 2019. Available at https://eprint.iacr.org/2019/499.pdf.

[OG19] Tobias Oder and Tim Güneysu. Implementing the NewHope-Simple key exchange on low-cost FPGAs. In Tanja Lange and Orr Dunkelman, editors, *Progress in Cryptology - LATINCRYPT 2017*, volume 11368 of *Lecture Notes in Computer Science*, pages 128–142. Springer, 2019.

[Pet17] Christophe Petit. Faster Algorithms for Isogeny Problems Using Torsion Point Images. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology - ASIACRYPT 2017*, volume 10625 of *Lecture Notes in Computer Science*, pages 330–353. Springer, 2017.

[PK16] Thomas B. Preußer and Markus Krause. Survey on and re-evaluation of wide adder architectures on FPGAs. In *International Conference on ReConFigurable Computing and FPGAs, ReConFig 2016, Cancun, Mexico, November 30 - Dec. 2, 2016*, pages 1–6, 2016. https://doi.org/10.1109/ReConFig.2016.7857189.

[PSW02] Matthew R. Pillmeier, Michael J. Schulte, and E. George Walters. Design alternatives for barrel shifters. *Proceedings of SPIE - The International Society for Optical Engineering*, 4791:436–447, 12 2002. https://doi.org/10.1117/12.452034.

[Ren18] Joost Renes. Computing Isogenies Between Montgomery Curves Using the Action of $(0, 0)$. In *PQCrypto*, volume 10786 of *Lecture Notes in Computer Science*, pages 229–247. Springer, 2018. https://ia.cr/2017/1198.

[RM19] Debapriya Basu Roy and Debdeep Mukhopadhyay. Post Quantum ECC on FPGA Platform. Cryptology ePrint Archive, Report 2019/568, 2019. https://eprint.iacr.org/2019/568.

[RMIT14] D. B. Roy, D. Mukhopadhyay, M. Izumi, and J. Takahashi. Tile before multiplication: An efficient strategy to optimize DSP multiplier for accelerating prime field ECC for NIST curves. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2014.

[Sco17] Michael Scott. Slothful reduction. Cryptology ePrint Archive, Report 2017/437, 2017. https://eprint.iacr.org/2017/437.

[SG17] P. Sasdrich and T. Güneysu. Cryptography for next generation TLS: Implementing the RFC 7748 elliptic Curve448 cryptosystem in hardware. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2017.

[Sil09] Joseph H. Silverman. *The Arithmetic of Elliptic Curves*. Graduate Texts in Mathematics. Springer-Verlag New York, 2009.

[SJA19] Hwajeong Seo, Amir Jalali, and Reza Azarderakhsh. SIKE Round 2 Speed Record on ARM Cortex-M4, 2019. Available at https://eprint.iacr.org/2019/535.pdf.

[SLLH18] Hwajeong Seo, Zhe Liu, Patrick Longa, and Zhi Hu. SIDH on ARM: Faster Modular Multiplications for Faster Post-Quantum Supersingular Isogeny Key Exchange. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2018(3):1–20, 2018. https://ia.cr/2018/700.

[The17] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 8.1)*, 2017. http://www.sagemath.org.

[TT03] A. F. Tenca and L. A. Tawalbeh. An efficient and scalable radix-4 modular multiplier design using recoding techniques. In *The Thrity-Seventh Asilomar Conference on Signals, Systems Computers, 2003*, volume 2, pages 1445–1450 Vol.2, Nov 2003.

[YAJ$^+$17] Youngho Yoo, Reza Azarderakhsh, Amir Jalali, David Jao, and Vladimir Soukharev. A Post-quantum Digital Signature Scheme Based on Supersingular Isogenies. In Aggelos Kiayias, editor, *Financial Cryptography and Data Security (FC 2017)*, volume 10322 of *Lecture Notes in Computer Science*, pages 163–181. Springer, 2017.

[ZSJP$^+$18] Gustavo H. M. Zanon, Marcos A. Simplicio Jr., Geovandro C. C. F. Pereira, Javad Doliskani, and Paulo S. L. M. Barreto. Faster Isogeny-Based Compressed Key Agreement. In Tanja Lange and Rainer Steinwandt, editors, *Post-Quantum Cryptography*, pages 248–268, Cham, 2018. Springer International Publishing. https://doi.org/10.1007/978-3-319-79063-3_12.

Appendix

	main flag	main append	main internal	operand o		operand b				operand a			
name	63 62 00	61 0	60 … 55 type	54 Dir	53 … 38 memo	37 Sign	36 Dir	35 Cx	34 … 19 memb	18 Sign	17 Dir	16 Cx	15 … 0 mema
nop	00	0	000000	0	0	0	0	0	0	0	0	0	0
jump	00	0	000001	0	memo	0	0	0	0	0	0	0	0
jumpeq	00	0	000001	0	memo	0	0	Cx	memb	0	0	Cx	mema
jumpl	00	0	000010	0	memo	0	0	Cx	memb	0	0	Cx	mema
jumpls	00	0	000010	0	memo	1	0	Cx	memb	1	0	Cx	mema
jumpeql	00	0	000011	0	memo	0	0	Cx	memb	0	0	Cx	mema
jumpeqls	00	0	000011	0	memo	1	0	Cx	memb	1	0	Cx	mema
push	00	0	000100	0	0	0	0	0	0	0	Dir	Cx	mema
pop	00	0	000101	Dir	memo	0	0	0	0	0	0	0	0
pushf	00	0	000110	0	0	0	0	0	0	0	Dir	1	mema
popf	00	0	000111	Dir	memo	0	0	0	0	0	0	0	0
pushm	00	0	001000	0	0	0	0	0	0	0	Dir	1	mema
popm	00	0	001001	Dir	memo	0	0	0	0	0	0	0	0
copy	00	0	001010	Dir	memo	0	0	0	0	0	Dir	Cx	mema
copyf	00	0	001011	Dir	memo	0	0	0	0	0	Dir	1	mema
copym	00	0	001100	Dir	memo	0	0	0	0	0	Dir	1	mema
lconstf	00	0	001101	Dir	memo	0	0	0	0	Sign	0	0	value
lconstm	00	0	001110	Dir	memo	0	0	0	0	Sign	0	0	value
call	00	0	001111	0	memo	0	0	0	0	0	0	1	pc
ret	00	0	010000	0	0	0	0	0	0	0	0	1	0
keccak_init	00	0	010010	0	0	0	0	0	0	0	0	0	0
keccak_go	00	0	010011	0	0	0	0	0	0	0	0	0	0
copya (inc)	00	0	0110 | Cd | Cs	Dir	memo	0	0	0	size	0	Dir	1	mema
copya (dec)	00	0	0111 | Cd | Cs	Dir	memo	0	0	0	size	0	Dir	1	mema
badd	00	0	100000	0	memo	0	0	Cx	memb	0	0	Cx	mema
bsub	00	0	100001	0	memo	0	0	Cx	memb	0	0	Cx	mema
bsmul	00	0	100010	0	memo	0	0	Cx	memb	0	0	Cx	mema
bsmuls	00	0	100010	0	memo	1	0	Cx	memb	1	0	Cx	mema
bshiftr	00	0	100100	0	memo	0	0	Cx	memb	0	0	Cx	mema
brotr	00	0	100101	0	memo	0	0	Cx	memb	0	0	Cx	mema
bshiftl	00	0	100110	0	memo	0	0	Cx	memb	0	0	Cx	mema
brotl	00	0	100111	0	memo	0	0	Cx	memb	0	0	Cx	mema
bland	00	0	101000	0	memo	0	0	Cx	memb	0	0	Cx	mema
blor	00	0	101001	0	memo	0	0	Cx	memb	0	0	Cx	mema
blxor	00	0	101010	0	memo	0	0	Cx	memb	0	0	Cx	mema
blnot	00	0	101010	0	memo	0	0	Cx	memb	0	0	0	1 … 1
fin	00	0	111111	0	0	0	0	0	0	0	0	0	0

Figure 17: The main instructions for the SIKE processor.

	carmela flag	carmela append	carmela internal	output o		input b				input a			
name	63 62 01	61 ... 59 000	58 ... 55 type	54 Dir	53 ... 38 Mo	37 0	36 Dir	35 En	34 ... 19 Mb	18 Sign	17 Dir	16 En	15 ... 0 Ma
mmuld	01	000	0000	Dir	Mo	0	Dir	En	Mb	0	Dir	En	Ma
msqud	01	000	0001	Dir	Mo	0	Dir	En	Ma	0	Dir	En	Ma
mmulm	01	000	0010	Dir	Mo	0	Dir	En	Mb	0	Dir	En	Ma
msqum	01	000	0011	Dir	Mo	0	Dir	En	Ma	0	Dir	En	Ma
madd_subd	01	000	0100	Dir	Mo	0	Dir	En	Mb	Sign	Dir	En	Ma
mitred	01	000	0101	Dir	Mo	0	0	0	0	0	Dir	En	Ma

Figure 18: The Carmela coprocessor instructions for the SIKE processor.